4 Handling requests with the middleware pipeline

This chapter covers

- Understanding middleware
- Serving static files using middleware
- Adding functionality using middleware
- Combining middleware to form a pipeline
- Handling exceptions and errors with middleware

In chapter 3 you had a whistle-stop tour of a complete ASP.NET Core application to see how the components come together to create a web application. In this chapter, we'll focus on one small subsection: the middleware pipeline.

In ASP.NET Core, *middleware* consists of C# classes or functions that handle an HTTP request or response. Middleware is chained together, with the output of one acting as the input to the next to form a pipeline.

The middleware pipeline is one of the most important parts of configuration for defining how your application behaves and how it responds to requests. Understanding how to build and compose middleware is key to adding functionality to your applications.

In this chapter you'll learn what middleware is and how to use it to create a pipeline. You'll see how you can chain multiple middleware components together,

with each component adding a discrete piece of functionality. The examples in this chapter are limited to using existing middleware components, showing how to arrange them in the correct way for your application. In chapter 31 you'll learn how to build your own middleware components and incorporate them into the pipeline.

We'll begin by looking at the concept of middleware, all the things you can achieve with it, and how a middleware component often maps to a cross-cutting concern. These functions of an application cut across multiple different layers. Logging, error handling, and security are classic cross-cutting concerns that are required by many parts of your application. Because all requests pass through the middleware pipeline, it's the preferred location to configure and handle this functionality.

In section 4.2 I'll explain how you can compose individual middleware components into a pipeline. You'll start out small, with a web app that displays only a holding page. From there, you'll learn how to build a simple static-file server that returns requested files from a folder on disk.

Next, you'll move on to a more complex pipeline containing multiple middleware. In this example you'll explore the importance of ordering in the middleware pipeline, and you'll see how requests are handled when your pipeline contains multiple middleware.

In section 4.3 you'll learn how you can use middleware to deal with an important aspect of any application: error handling. Errors are a fact of life for all applications, so it's important that you account for them when building your app.

You can handle errors in a few ways. Errors are among the classic cross-cutting concerns, and middleware is well placed to provide the required functionality. In section 4.3 I'll show how you can handle exceptions with middleware provided by Microsoft. In particular, you'll learn about two different components:

- `DeveloperExceptionPageMiddleware`—Provides quick error feedback when building an application
- `ExceptionHandlerMiddleware`—Provides a generic error page in production so that you don't leak any sensitive details

You won't see how to build your own middleware in this chapter; instead, you'll see that you can go a long way by using the components provided as part of ASP.NET Core. When you understand the middleware pipeline and its behavior, you'll find it much easier to understand when and why custom middleware is required. With that in mind, let's dive in!

4.1 Defining middleware

The word *middleware* is used in a variety of contexts in software development and IT, but it's not a particularly descriptive word.

In ASP.NET Core, *middleware* is a C# class[1] that can handle an HTTP request or response. Middleware can

[1] Technically, middleware needs to be a *function*, as you'll see in chapter 31, but it's common to implement middleware as a C# class with a single method.

- Handle an incoming HTTP *request* by generating an HTTP *response*
- Process an incoming HTTP *request*, modify it, and pass it on to another piece of middleware
- Process an outgoing HTTP *response*, modify it, and pass it on to another piece of middleware or to the ASP.NET Core web server

You can use middleware in a multitude of ways in your own applications. A piece of logging middleware, for example, might note when a request arrived and then pass it on to another piece of middleware. Meanwhile, a static-file middleware component might spot an incoming request for an image with a specific name, load the image from disk, and send it back to the user without passing it on.

The most important piece of middleware in most ASP.NET Core applications is the `EndpointMiddleware` class. This class normally generates all your HTML and JavaScript Object Notation (JSON) responses, and is the focus of most of this book. Like image-resizing middleware, it typically receives a request, generates a response, and then sends it back to the user (figure 4.1).

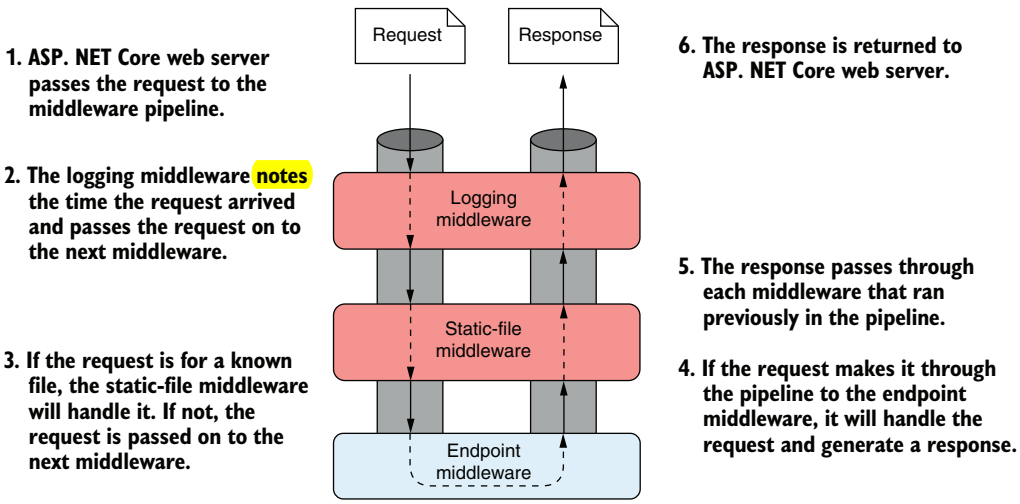


Figure 4.1 Example of a middleware pipeline. Each middleware component handles the request and passes it on to the next middleware component in the pipeline. After a middleware component generates a response, it passes the response back through the pipeline. When it reaches the ASP.NET Core web server, the response is sent to the user's browser.

DEFINITION This arrangement—whereby a piece of middleware can call another piece of middleware, which in turn can call another, and so on—is referred to as a *pipeline*. You can think of each piece of middleware as being like a section of pipe; when you connect all the sections, a request flows through one piece and into the next.

One of the most common use cases for middleware is for the cross-cutting concerns of your application. These aspects of your application need to occur for every request, regardless of the specific path in the request or the resource requested, including

- Logging each request
- Adding standard security headers to the response
- Associating a request with the relevant user
- Setting the language for the current request

In each of these examples, the middleware receives a request, modifies it, and then passes the request on to the next piece of middleware in the pipeline. Subsequent middleware could use the details added by the earlier middleware to handle the request in some way. In figure 4.2, for example, the authentication middleware associates the request with a user. Then the authorization middleware uses this detail to verify whether the user has permission to make that specific request to the application.

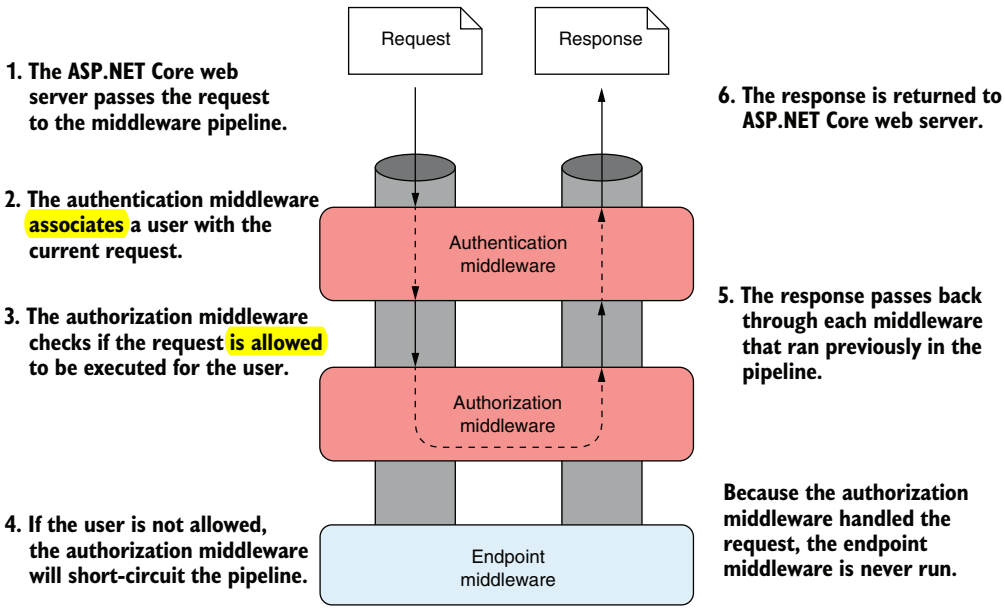


Figure 4.2 Example of a middleware component modifying a request for use later in the pipeline. Middleware can also short-circuit the pipeline, returning a response before the request reaches later middleware.

If the user has permission, the authorization middleware passes the request on to the endpoint middleware to allow it to generate a response. If the user doesn't have permission, the authorization middleware can short-circuit the pipeline, generating a response directly; it returns the response to the previous middleware, and the endpoint middleware never sees the request. This scenario is an example of the *chain-of-responsibility* design pattern.

DEFINITION When a middleware component short-circuits the pipeline and returns a response, it's called *terminal middleware*.

A key point to glean from this example is that the pipeline is *bidirectional*. The request passes through the pipeline in one direction until a piece of middleware generates a response, at which point the response passes *back* through the pipeline, passing through each piece of middleware a *second* time, in reverse order, until it gets back to the first piece of middleware. Finally, the first/last piece of middleware passes the response back to the ASP.NET Core web server.

The `HttpContext` object

I mentioned the `HttpContext` in chapter 3, and it's sitting behind the scenes here, too. The ASP.NET Core web server constructs an `HttpContext` for each request, which the ASP.NET Core application uses as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with and stored in it. Examples are properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details, and then passes it on to the middleware pipeline and the rest of the application.

All middleware has access to the `HttpContext` for a request. It can use this object to determine whether the request contains any user credentials, to identify which page the request is attempting to access, and to fetch any posted data, for example. Then it can use these details to determine how to handle the request.

When the application finishes processing the request, it updates the `HttpContext` with an appropriate response and returns it through the middleware pipeline to the web server. Then the ASP.NET Core web server converts the representation to a raw HTTP response and sends it back to the reverse proxy, which forwards it to the user's browser.

As you saw in chapter 3, you define the middleware pipeline in code as part of your initial application configuration in Program.cs. You can tailor the middleware pipeline specifically to your needs; simple apps may need only a short pipeline, whereas large apps with a variety of features may use much more middleware. Middleware is the fundamental source of behavior in your application. Ultimately, the middleware pipeline is responsible for responding to any HTTP requests it receives.

Requests are passed to the middleware pipeline as `HttpContext` objects. As you saw in chapter 3, the ASP.NET Core web server builds an `HttpContext` object from an incoming request, which passes up and down the middleware pipeline. When you're using existing middleware to build a pipeline, this detail is one that you'll rarely have to deal with. But as you'll see in the final section of this chapter, its presence behind the scenes provides a route to exerting extra control over your middleware pipeline.

You can also think of your middleware pipeline as being a series of concentric components, similar to a traditional matryoshka (Russian) doll, as shown in figure 4.3.

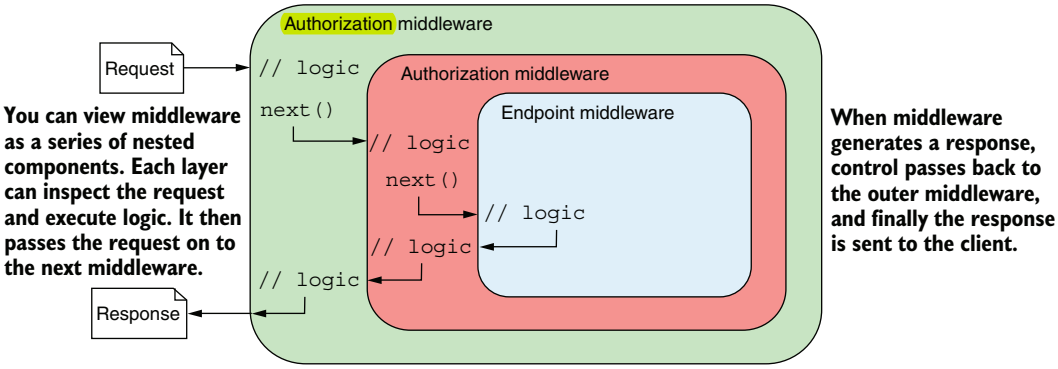


Figure 4.3 You can also think of middleware as being a series of nested components; a request is sent deeper into the middleware, and the response resurfaces from it. Each middleware component can execute logic before passing the response on to the next middleware component and can execute logic after the response has been created, on the way back out of the stack.

A request progresses through the pipeline by heading deeper into the stack of middleware until a response is returned. Then the response returns through the middleware, passing through the components in reverse order from the request.

Middleware vs. HTTP modules and HTTP handlers

In the previous version of ASP.NET, the concept of a middleware pipeline isn't used. Instead, you have HTTP modules and HTTP handlers.

An *HTTP handler* is a process that runs in response to a request and generates the response. The ASP.NET page handler, for example, runs in response to requests for .aspx pages. Alternatively, you could write a custom handler that returns resized images when an image is requested.

HTTP modules handle the cross-cutting concerns of applications, such as security, logging, and session management. They run in response to the life-cycle events that a request progresses through when it's received by the server. Examples of events include `BeginRequest`, `AcquireRequestState`, and `PostAcquireRequestState`.

This approach works, but sometimes it's tricky to reason about which modules will run at which points. Implementing a module requires relatively detailed understanding of the state of the request at each individual life-cycle event.

The middleware pipeline makes understanding your application far simpler. The pipeline is defined completely in code, specifying which components should run and in which order. Behind the scenes, the middleware pipeline in ASP.NET Core is simply a chain of method calls, with each middleware function calling the next in the pipeline.

That's pretty much all there is to the concept of middleware. In the next section, I'll discuss ways you can combine middleware components to create an application and how to use middleware to separate the concerns of your application.

4.2 Combining middleware in a pipeline

Generally speaking, each middleware component has a single primary concern; it handles only one aspect of a request. Logging middleware deals only with logging the request, authentication middleware is concerned only with identifying the current user, and static-file middleware is concerned only with returning static files.

Each of these concerns is highly focused, which makes the components themselves small and easy to reason about. This approach also gives your app added flexibility. Adding static-file middleware, for example, doesn't mean you're forced to have image-resizing behavior or authentication; each of these features is an additional piece of middleware.

To build a complete application, you compose multiple middleware components into a pipeline, as shown in section 4.1. Each middleware component has access to the original request, as well as any changes made to the `HttpContext` by middleware earlier in the pipeline. When a response has been generated, each middleware component can inspect and/or modify the response as it passes back through the pipeline before it's sent to the user. This feature allows you to build complex application behaviors from small, focused components.

In the rest of this section, you'll see how to create a middleware pipeline by combining various middleware components. Using standard middleware components, you'll learn to create a holding page and to serve static files from a folder on disk. Finally, you'll take a look at a more complex pipeline such as you'd get in a minimal API application with multiple middleware, routing, and endpoints.

4.2.1 Simple pipeline scenario 1: A holding page

For your first app in this chapter and your first middleware pipeline, you'll learn how to create an app consisting of a holding page. Adding a holding page can be useful occasionally when you're setting up your application to ensure that it's processing requests without errors.

> **TIP** Remember that you can view the application code for this book in the GitHub repository at http://mng.bz/Y1qN.

In previous chapters, I mentioned that the ASP.NET Core framework is composed of many small individual libraries. You typically add a piece of middleware by referencing a package in your application's .csproj project file and configuring the middleware in Program.cs. Microsoft ships many standard middleware components with ASP.NET Core for you to choose among; you can also use third-party components from NuGet and GitHub, or you can build your own custom middleware. You can find the list of built-in middleware at http://mng.bz/Gyxq.

> **NOTE** I discuss building custom middleware in chapter 31.

In this section, you'll see how to create one of the simplest middleware pipelines, consisting only of `WelcomePageMiddleware`. `WelcomePageMiddleware` is designed to provide a

sample HTML page quickly when you're first developing an application, as you can see in figure 4.4. You wouldn't use it in a production app, as you can't customize the output, but it's a single, self-contained middleware component you can use to ensure that your application is running correctly.

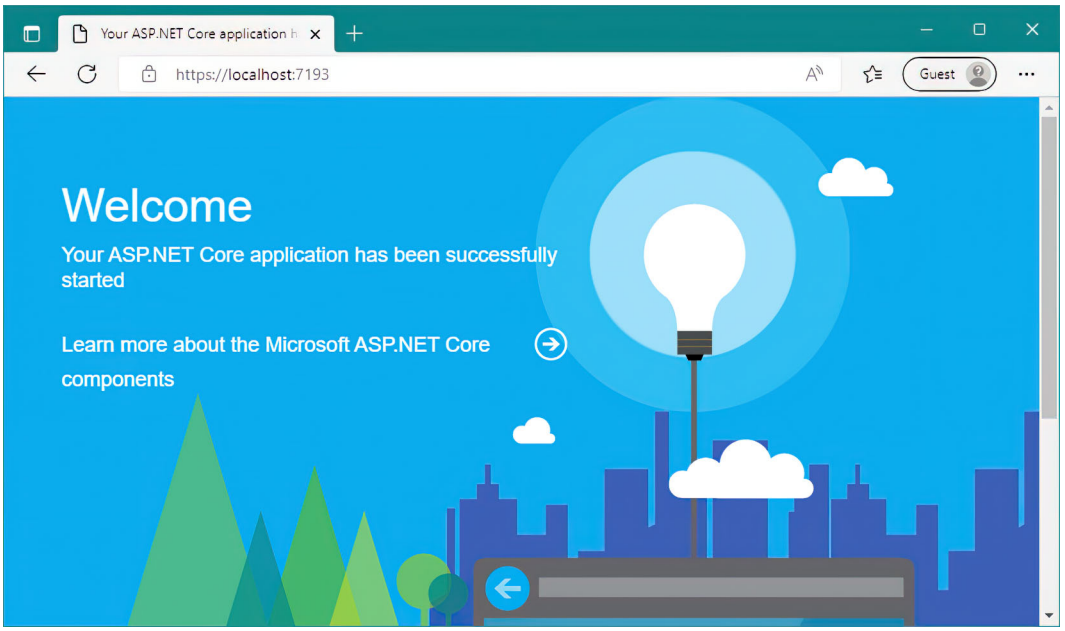


Figure 4.4 The Welcome-page middleware response. Every request to the application, at any path, will return the same Welcome-page response.

> **TIP** `WelcomePageMiddleware` is included as part of the base ASP.NET Core framework, so you don't need to add a reference to any additional NuGet packages.

Even though this application is simple, the same process you've seen before occurs when the application receives an HTTP request, as shown in figure 4.5.

The request passes to the ASP.NET Core web server, which builds a representation of the request and passes it to the middleware pipeline. As it's the first (only!) middleware in the pipeline, `WelcomePageMiddleware` receives the request and must decide how to handle it. The middleware responds by generating an HTML response, no matter what request it receives. This response passes back to the ASP.NET Core web server, which forwards it to the reverse proxy and then to the user to display in their browser.

As with all ASP.NET Core applications, you define the middleware pipeline in Program.cs by calling `Use*` methods on the `WebApplication` instance. To create your first middleware pipeline, which consists of a single middleware component, you need a

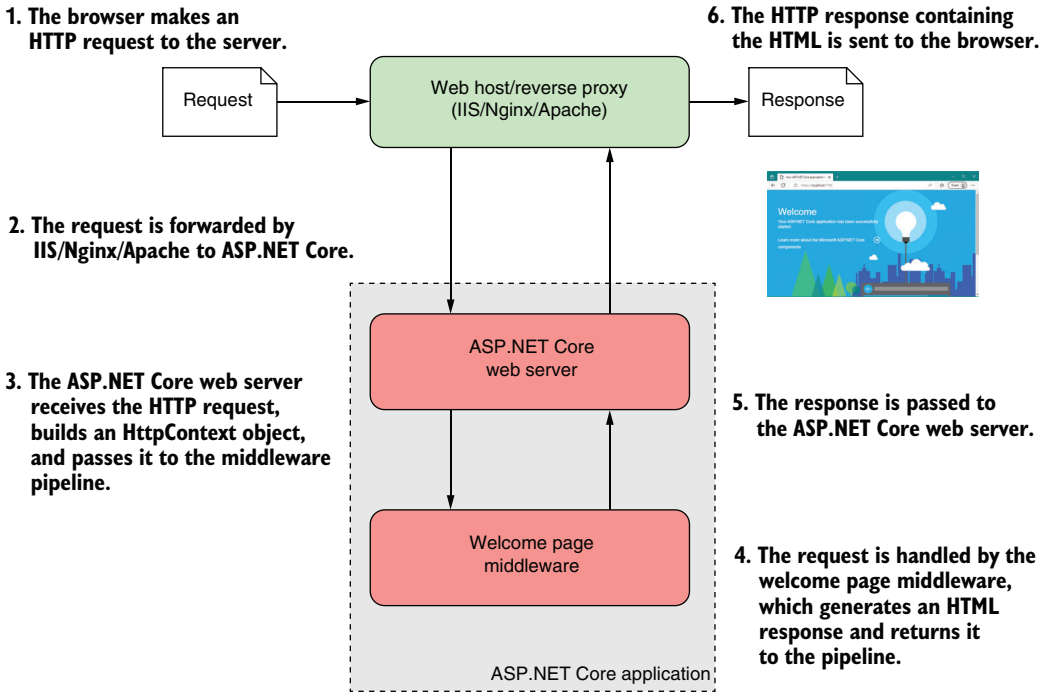


Figure 4.5 `WelcomePageMiddleware` handles a request. The request passes from the reverse proxy to the ASP.NET Core web server and finally to the middleware pipeline, which generates an HTML response.

single method call. The application doesn't need any extra configuration or services, so your whole application consists of the four lines in the following listing.

Listing 4.1 Program.cs for a Welcome-page middleware pipeline

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseWelcomePage();

app.Run();
```

Uses the default WebApplication configuration

The only custom middleware in the pipeline

Runs the application to handle requests

You build up the middleware pipeline in ASP.NET Core by calling methods on `WebApplication` (which implements `IApplicationBuilder`). `WebApplication` doesn't define methods like `UseWelcomePage` itself; instead, these are *extension* methods.

Using extension methods allows you to add functionality to the `WebApplication` class, while keeping the implementation isolated from it. Under the hood, the methods typically call *another* extension method to add the middleware to the pipeline. Behind the scenes, for example, the `UseWelcomePage` method adds the `WelcomePageMiddleware` to the pipeline by calling

```
UseMiddleware<WelcomePageMiddleware>();
```

This convention of creating an extension method for each piece of middleware and starting the method name with `Use` is designed to improve discoverability when you add middleware to your application.[2] ASP.NET Core includes a lot of middleware as part of the core framework, so you can use IntelliSense in Visual Studio and other integrated development environments (IDEs) to view all the middleware that's available, as shown in figure 4.6.

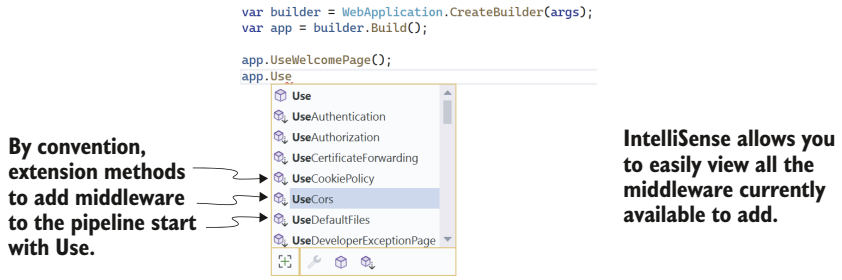


Figure 4.6 IntelliSense makes it easy to view all the available middleware to add to your middleware pipeline.

Calling the `UseWelcomePage` method adds the `WelcomePageMiddleware` as the next middleware in the pipeline. Although you're using only a single middleware component here, it's important to remember that the order in which you make calls to `IApplicationBuilder` in `Configure` defines the order in which the middleware will run in the pipeline.

> **WARNING** When you're adding middleware to the pipeline, always take care to consider the order in which it will run. A component can access only data created by middleware that comes before it in the pipeline.

This application is the most basic kind, returning the same response no matter which URL you navigate to, but it shows how easy it is to define your application behavior with middleware. Next, we'll make things a little more interesting by returning different responses when you make requests to different paths.

4.2.2 Simple pipeline scenario 2: Handling static files

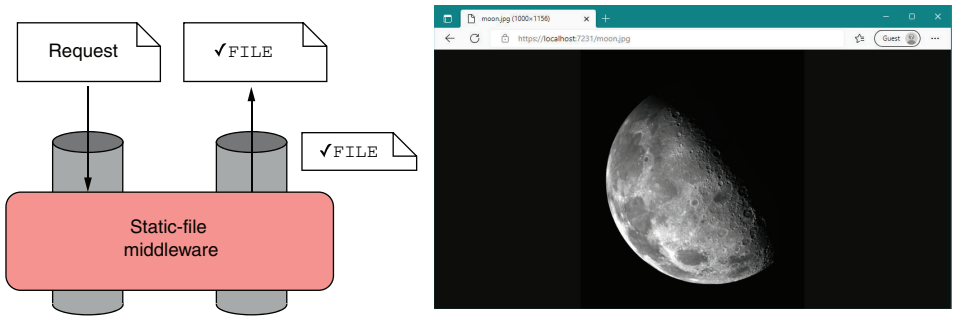
In this section, I'll show you how to create one of the simplest middleware pipelines you can use for a full application: a static-file application. Most web applications, including those with dynamic content, serve some pages by using static files. Images, JavaScript, and CSS stylesheets are normally saved to disk during development and are served up when requested from the special wwwroot folder of your project, normally as part of a full HTML page request.

---

[2] The downside to this approach is that it can hide exactly which middleware is being added to the pipeline. When the answer isn't clear, I typically search for the source code of the extension method directly in GitHub (https://github.com/aspnet/aspnetcore).

DEFINITION By default, the wwwroot folder is the only folder in your application that ASP.NET Core will serve files from. It doesn't serve files from other folders for security reasons. The wwwroot folder in an ASP.NET Core project is typically deployed as is to production, including all the files and folders it contains.

You can use `StaticFileMiddleware` to serve static files from the wwwroot folder when requested, as shown in figure 4.7. In this example, an image called moon.jpg exists in the wwwroot folder. When you request the file using the `/moon.jpg` path, it's loaded and returned as the response to the request.



1. The static-file middleware handles the request by returning the file requested.
2. The file stream is sent back through the middleware pipeline and out to the browser.
3. The browser displays the file returned in the response.

Figure 4.7 Serving a static image file using the static-file middleware

If the user requests a file that doesn't exist in the wwwroot folder, such as missing.jpg, the static-file middleware won't serve a file. Instead, a 404 HTTP error code response will be sent to the user's browser, which displays its default "File Not Found" page, as shown in figure 4.8.

NOTE How this page looks depends on your browser. In some browsers, you may see a blank page.

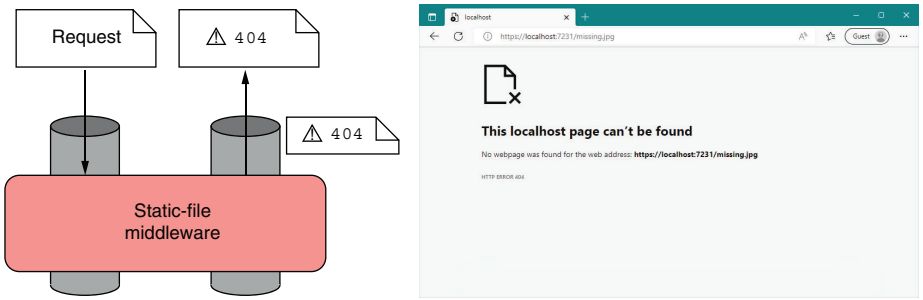
Building the middleware pipeline for this simple static-file application is easy. The pipeline consists of a single piece of middleware, `StaticFileMiddleware`, as you can see in the following listing. You don't need any services, so configuring the middleware pipeline with `UseStaticFiles` is all that's required.

Listing 4.2 Program.cs for a static-file middleware pipeline

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseStaticFiles();   <-- Adds the StaticFileMiddleware to the pipeline

app.Run();
```



1. The static-file middleware handles the request by trying to return the requested file, but as it doesn't exist, it returns a raw 404 response.

2. The 404 HTTP error code is sent back through the middleware pipeline to the user.

3. The browser displays its default "File Not Found" error page.

Figure 4.8 Returning a 404 to the browser when a file doesn't exist. The requested file didn't exist in the wwwroot folder, so the ASP.NET Core application returned a 404 response. Then the browser (Microsoft Edge, in this case) shows the user a default "File Not Found" error page.

TIP Remember that you can view the application code for this book in the GitHub repository at http://mng.bz/Y1qN.

When the application receives a request, the ASP.NET Core web server handles it and passes it to the middleware pipeline. `StaticFileMiddleware` receives the request and determines whether it can handle it. If the requested file exists, the middleware handles the request and returns the file as the response, as shown in figure 4.9.

If the file doesn't exist, the request effectively passes *through* the static-file middleware unchanged. But wait—you added only one piece of middleware, right? Surely you can't pass the request through to the next middleware component if there *isn't* another one.

ASP.NET Core automatically adds a dummy piece of middleware to the end of the pipeline. This middleware always returns a 404 response if it's called.

TIP If no middleware generates a response for a request, the pipeline automatically returns a simple 404 error response to the browser.

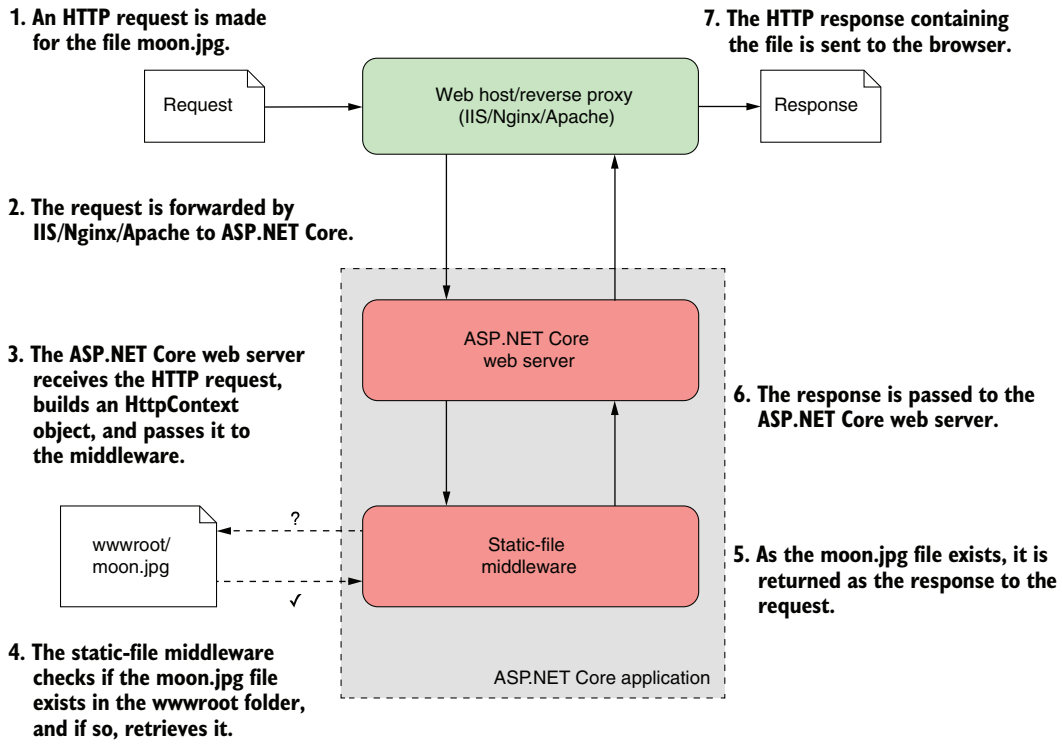


Figure 4.9 `StaticFileMiddleware` **handles a request for a file. The middleware checks the wwwroot folder to see if whether requested moon.jpg file exists. The file exists, so the middleware retrieves it and returns it as the response to the web server and, ultimately, to the browser.**

HTTP response status codes

Every HTTP response contains a *status code* and, optionally, a *reason phrase* describing the status code. Status codes are fundamental to the HTTP protocol and are a standardized way of indicating common results. A 200 response, for example, means that the request was successfully answered, whereas a 404 response indicates that the resource requested couldn't be found. You can see the full list of standardized status codes at https://www.rfc-editor.org/rfc/rfc9110#name-status-codes.

Status codes are always three digits long and are grouped in five classes, based on the first digit:

- *1xx*—Information. This code is not often used; it provides a general acknowledgment.
- *2xx*—Success. The request was successfully handled and processed.
- *3xx*—Redirection. The browser must follow the provided link to allow the user to log in, for example.

(continued)

- *4xx*—Client error. A problem occurred with the request. The request sent invalid data, for example, or the user isn't authorized to perform the request.
- *5xx*—Server error. A problem on the server caused the request to fail.

These status codes typically drive the behavior of a user's browser. The browser will handle a 301 response automatically, for example, by redirecting to the provided new link and making a second request, all without the user's interaction.

Error codes are in the 4xx and 5xx classes. Common codes include a 404 response when a file couldn't be found, a 400 error when a client sends invalid data (such as an invalid email address), and a 500 error when an error occurs on the server. HTTP responses for error codes may include a response body, which is content to display when the client receives the response.

This basic ASP.NET Core application makes it easy to see the behavior of the ASP.NET Core middleware pipeline and the static-file middleware in particular, but it's unlikely that your applications will be this simple. It's more likely that static files will form one part of your middleware pipeline. In the next section you'll see how to combine multiple middleware components as we look at a simple minimal API application.

4.2.3 Simple pipeline scenario 3: A minimal API application

By this point, you should have a decent grasp of the middleware pipeline, insofar as you understand that it defines your application's behavior. In this section you'll see how to combine several standard middleware components to form a pipeline. As before, you do this in Program.cs by adding middleware to the `WebApplication` object.

You'll begin by creating a basic middleware pipeline that you'd find in a typical ASP.NET Core minimal APIs template and then extend it by adding middleware. Figure 4.10 shows the output you see when you navigate to the home page of the application—identical to the sample application in chapter 3.

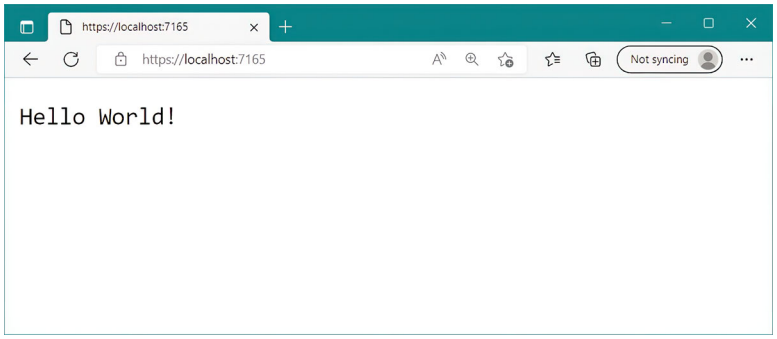


Figure 4.10 A simple minimal API application. The application uses only four pieces of middleware: routing middleware to choose the endpoint to run, endpoint middleware to generate the response from a Razor Page, static-file middleware to serve image files, and exception-handler middleware to capture any errors.

Creating this application requires only four pieces of middleware: routing middleware to choose a minimal API endpoint to execute, endpoint middleware to generate the response, static-file middleware to serve any image files from the wwwroot folder, and exception-handler middleware to handle any errors that might occur. Even though this example is still a `Hello World!` example, this architecture is much closer to a realistic example. The following listing shows an example of such an application.

Listing 4.3 A basic middleware pipeline for a minimal APIs application

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseDeveloperExceptionPage();
app.UseStaticFiles();
app.UseRouting();

app.MapGet("/", () => "Hello World!");

app.Run();
```

This call isn't strictly necessary, as it's already added by WebApplication by default.

Adds the RoutingMiddleware to the pipeline

Defines an endpoint for the application

Adds the StaticFileMiddleware to the pipeline

The addition of middleware to `WebApplication` to form the pipeline should be familiar to you now, but several points are worth noting in this example:

- Middleware is added with `Use*()` methods.
- `MapGet` defines an *endpoint*, not middleware. It defines the endpoints that the routing and endpoint middleware can use.
- `WebApplication` automatically adds some middleware to the pipeline, such as the `EndpointMiddleware`.
- The order of the `Use*()` method calls is important and defines the order of the middleware pipeline.

First, all the methods for adding middleware start with `Use`. As I mentioned earlier, this is thanks to the convention of using extension methods to extend the functionality of `WebApplication`; prefixing the methods with `Use` should make them easier to discover.

Second, it's important to understand that the `MapGet` method does *not* add middleware to the pipeline; it defines an *endpoint* in your application. These endpoints are used by the routing and endpoint middleware. You'll learn more about endpoints and routing in chapter 5.

> **TIP** You can define the endpoints for your app by using `MapGet()` anywhere in Program.cs before the call to `app.Run()`, but the calls are typically placed after the middleware pipeline definition.

In chapter 3, I mentioned that `WebApplication` automatically adds middleware to your app. You can see this process in action in listing 4.3 automatically adding the `EndpointMiddleware` to the end of the middleware pipeline. `WebApplication` also

automatically adds the developer exception page middleware to the *start* of the middleware pipeline when you're running in development. As a result, you can omit the call to `UseDeveloperExceptionPage()` from listing 4.3, and your middleware pipeline will be essentially the same.

WebApplication and autoadded middleware

`WebApplication` and `WebApplicationBuilder` were introduced in .NET 6 to try to reduce the amount of boilerplate code required for a `Hello World!` ASP.NET Core application. As part of this initiative, Microsoft chose to have `WebApplication` *automatically* add various middleware to the pipeline. This decision alleviates some of the common getting-started pain points of middleware ordering by ensuring that, for example, `UseRouting()` is always called before `UseAuthorization()`.

Everything has trade-offs, of course, and for `WebApplication` the trade-off is that it's harder to understand exactly what's in your middleware pipeline without having deep knowledge of the framework code itself.

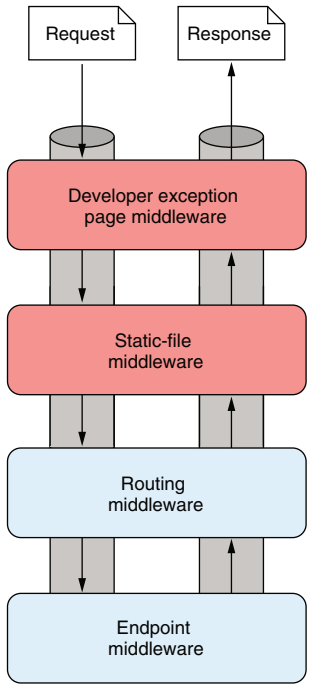
Luckily, you don't need to worry about the middleware that `WebApplication` adds for the most part. If you're new to ASP.NET Core, generally you can accept that `WebApplication` will add the middleware only when it's necessary and safe to do so.

Nevertheless, in some cases it may pay to know exactly what's in your pipeline, especially if you're familiar with ASP.NET Core. In .NET 7, `WebApplication` automatically adds some or all of the following middleware to the start of the middleware pipeline:

- `HostFilteringMiddleware`—This middleware is security-related. You can read more about why it's useful and how to configure it at http://mng.bz/zXxa.
- `ForwardedHeadersMiddleware`—This middleware controls how forwarded headers are handled. You can read more about it in chapter 27.
- `DeveloperExceptionPageMiddleware`—As already discussed, this middleware is added when you run in a development environment.
- `RoutingMiddleware`—If you add any endpoints to your application, `UseRouting()` runs before you add any custom middleware to your application.
- `AuthenticationMiddleware`—If you configure authentication, this middleware authenticates a user for the request. Chapter 23 discusses authentication in detail.
- `AuthorizationMiddleware`—The authorization middleware runs after authentication and determines whether a user is permitted to execute an endpoint. If the user doesn't have permission, the request is short-circuited. I discuss authorization in detail in chapter 24.
- `EndpointMiddleware`—This middleware pairs with the `RoutingMiddleware` to *execute* an endpoint. Unlike the other middleware described here, the `EndpointMiddleware` is added to the *end* of the middleware pipeline, after any other middleware you configure in Program.cs.

Depending on your Program.cs configuration, `WebApplication` may not add all this middleware. Also, if you don't want some of this automatic middleware to be at the start of your middleware pipeline, generally you can override the location. In listing 4.3, for example, we override the automatic `RoutingMiddleware` location by calling `UseRouting()` explicitly, ensuring that routing occurs exactly where we need it.

Another important point about listing 4.3 is that the order in which you add the middleware to the `WebApplication` object is the order in which the middleware is added to the pipeline. The order of the calls in listing 4.3 creates a pipeline similar to that shown in figure 4.11.



The developer exception page middleware was added first, so it is the first (and last) middleware to process the request.

The static-file middleware is the second middleware in the pipeline. It handles requests for static files before they get to the endpoint middleware.

The routing middleware attempts to find an endpoint that will handle the request.

The endpoint middleware is the last in the pipeline. If there is no endpoint to handle the request, the pipeline returns a 404 response.

Figure 4.11 The middleware pipeline for the example application in listing 4.3. The order in which you add the middleware to `WebApplication` defines the order of the middleware in the pipeline.

The ASP.NET Core web server passes the incoming request to the developer exception page middleware first. This exception-handler middleware ignores the request initially; its purpose is to catch any exceptions thrown by later middleware in the pipeline, as you'll see in section 4.3. It's important for this middleware to be placed early in the pipeline so that it can catch errors produced by later middleware.

The developer exception page middleware passes the request on to the static-file middleware. The static-file handler generates a response if the request corresponds to a file; otherwise, it passes the request on to the routing middleware. The routing middleware selects a minimal API endpoint based on the endpoints defined and the request URL, and the endpoint middleware executes the selected minimal API endpoint. If no endpoint can handle the requested URL, the automatic dummy middleware returns a 404 response.

In chapter 3, I mentioned that `WebApplication` adds the `RoutingMiddleware` to the start of the middleware pipeline automatically. So you may be wondering why I explicitly added it to the pipeline in listing 4.3 using `UseRouting()`.

The answer, again, is related to the order of the middleware. Adding an explicit call to `UseRouting()` tells `WebApplication` *not* to add the `RoutingMiddleware` automatically before the middleware defined in Program.cs. This allows us to "move" the `RoutingMiddleware` to be placed *after* the `StaticFileMiddleware`. Although this step isn't strictly necessary in this case, it's good practice. The `StaticFileMiddleware` doesn't use routing, so it's preferable to let this middleware check whether the incoming request is for a static file; if so, it can short-circuit the pipeline and avoid the unnecessary call to the `RoutingMiddleware`.

> **NOTE** In versions 1.x and 2.x of ASP.NET Core, the routing and endpoint middleware were combined in a single Model-View-Controller (MVC) middleware component. Splitting the responsibilities for routing from execution makes it possible to insert middleware *between* the routing and endpoint middleware. I discuss routing further in chapters 6 and 14.

The impact of ordering is most obvious when you have two pieces of middleware that are listening for the same path. The endpoint middleware in the example pipeline currently responds to a request to the home page of the application (with the `/` path) by returning the string `"Hello World!"`, as shown in figure 4.10. Figure 4.12 shows what happens if you reintroduce a piece of middleware that you saw previously, `WelcomePageMiddleware`, and configure it to respond to the `/` path as well.

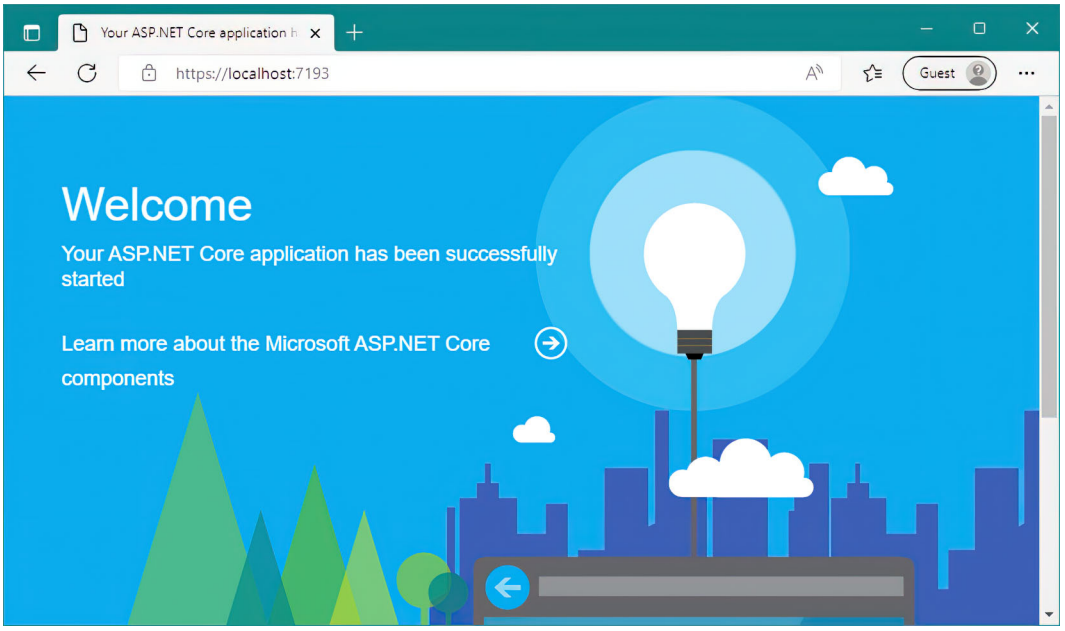


Figure 4.12 The Welcome-page middleware response. The Welcome-page middleware comes before the endpoint middleware, so a request to the home page returns the Welcome-page middleware instead of the minimal API response.

As you saw in section 4.2.1, `WelcomePageMiddleware` is designed to return a fixed HTML response, so you wouldn't use it in a production app, but it illustrates the point nicely. In the following listing, it's added to the start of the middleware pipeline and configured to respond only to the `"/"` path.

Listing 4.4 Adding `WelcomePageMiddleware` to the pipeline

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseWelcomePage("/");
app.UseDeveloperExceptionPage();
app.UseStaticFiles();
app.UseRouting();

app.MapGet("/", () => "Hello World!");

app.Run();
```

WelcomePageMiddleware handles all requests to the "/" path and returns a sample HTML response.

Requests to "/" will never reach the endpoint middleware, so this endpoint won't be called.

Even though you know that the endpoint middleware can *also* handle the `"/"` path, `WelcomePageMiddleware` is earlier in the pipeline, so it returns a response when it receives the request to `"/"`, short-circuiting the pipeline, as shown in figure 4.13. None of the other middleware in the pipeline runs for the request, so none has an opportunity to generate a response.

As `WebApplication` automatically adds `EndpointMiddleware` to the *end* of the middleware pipeline, the `WelcomePageMiddleware` will *always* be ahead of it, so it always generates a response before the endpoint can execute in this example.

> **TIP** You should always consider the order of middleware when adding it to `WebApplication`. Middleware added earlier in the pipeline will run (and potentially return a response) before middleware added later.

All the examples shown so far try to handle an incoming request and generate a response, but it's important to remember that the middleware pipeline is bidirectional. Each middleware component gets an opportunity to handle both the incoming request and the outgoing response. The order of middleware is most important for those components that create or modify the outgoing response.

In listing 4.3, I included `DeveloperExceptionPageMiddleware` at the start of the application's middleware pipeline, but it didn't seem to do anything. Error-handling middleware characteristically ignores the incoming request as it arrives in the pipeline; instead, it inspects the outgoing response, modifying it only when an error has occurred. In the next section, I discuss the types of error-handling middleware that are available to use with your application and when to use them.

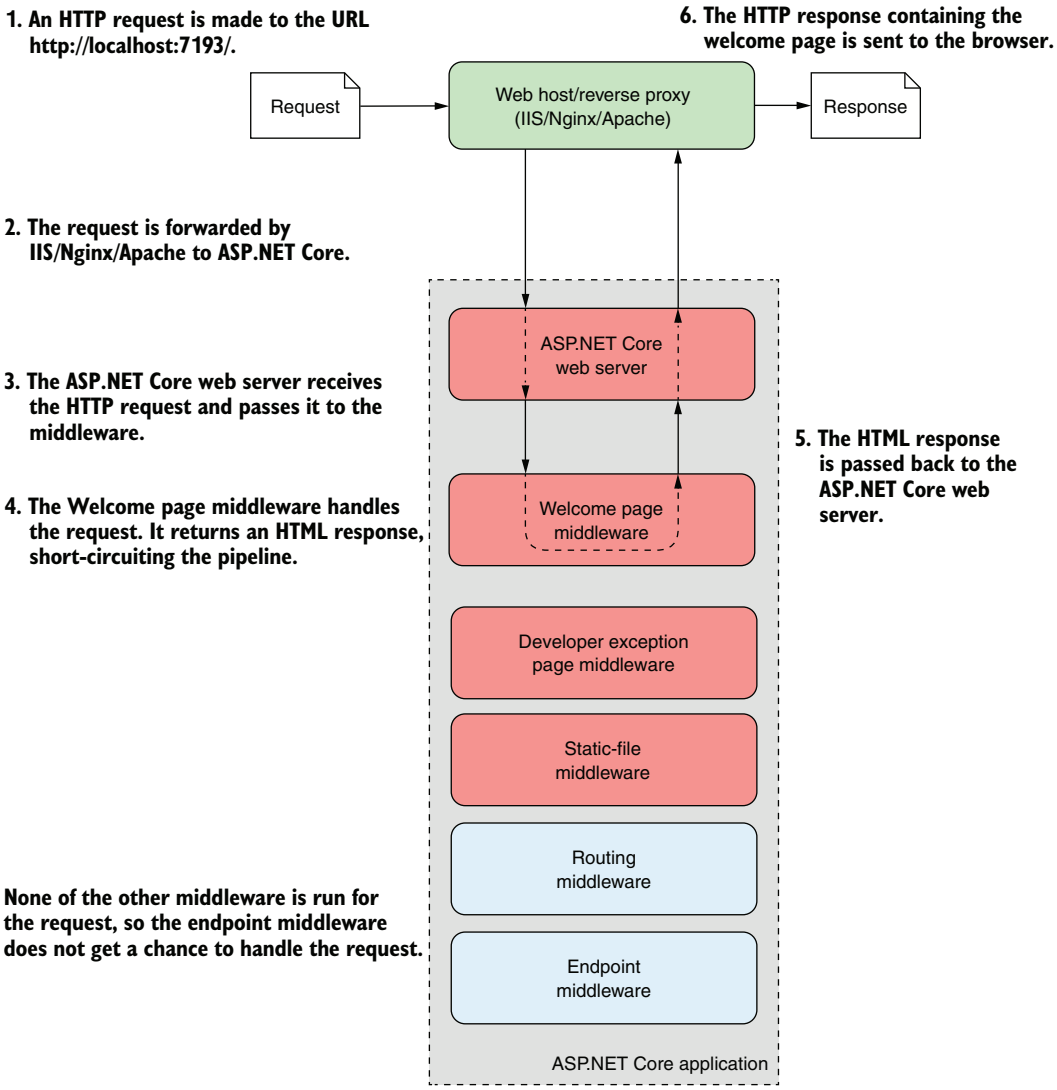


Figure 4.13 Overview of the application handling a request to the "/" path. The Welcome-page middleware is first in the middleware pipeline, so it receives the request before any other middleware. It generates an HTML response, short-circuiting the pipeline. No other middleware runs for the request.

4.3 Handling errors using middleware

Errors are a fact of life when you're developing applications. Even if you write perfect code, as soon as you release and deploy your application, users will find a way to break it, by accident or intentionally! The important thing is that your application handles these errors gracefully, providing a suitable response to the user and not causing your whole application to fail.

The design philosophy for ASP.NET Core is that every feature is opt-in. So because error handling is a feature, you need to enable it explicitly in your application. Many types of errors could occur in your application, and you have many ways to handle them, but in this section I focus on a single type of error: exceptions.

Exceptions typically occur whenever you find an unexpected circumstance. A typical (and highly frustrating) exception you'll no doubt have experienced before is `NullReferenceException`, which is thrown when you attempt to access a variable that hasn't been initialized.[3] If an exception occurs in a middleware component, it propagates up the pipeline, as shown in figure 4.14. If the pipeline doesn't handle the exception, the web server returns a 500 status code to the user.

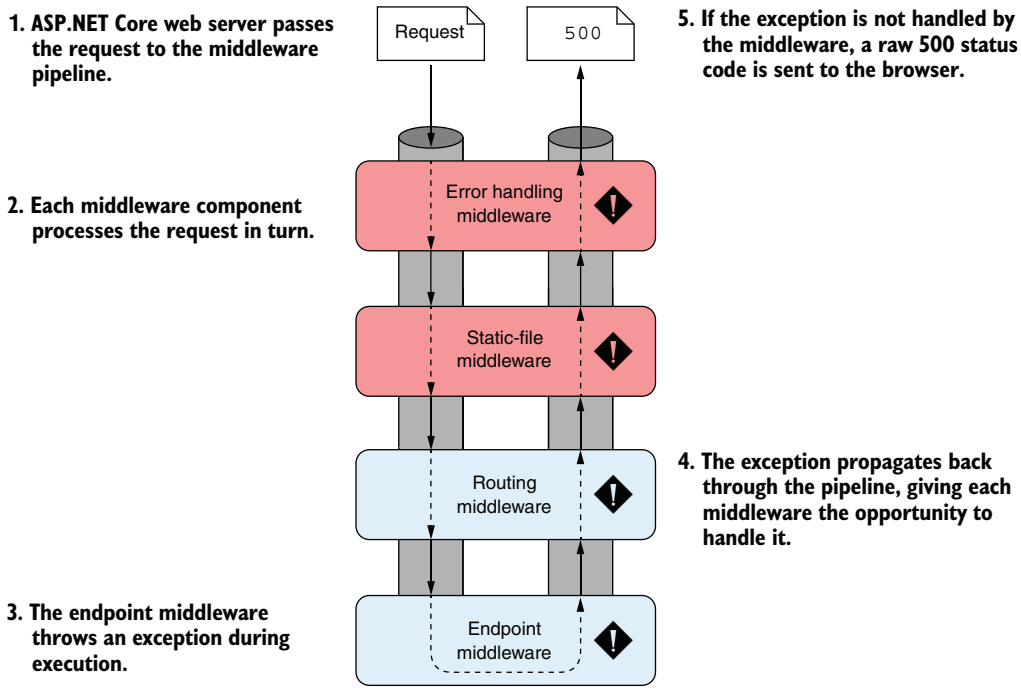


Figure 4.14 An exception in the endpoint middleware propagates through the pipeline. If the exception isn't caught by middleware earlier in the pipeline, a 500 "Server error" status code is sent to the user's browser.

In some situations, an error won't cause an exception. Instead, middleware might generate an error status code. One such case occurs when a requested path isn't handled. In that situation, the pipeline returns a 404 error.

[3] C# 8.0 introduced non-nullable reference types, which provide a way to handle null values more clearly, with the promise of finally ridding .NET of `NullReferenceExceptions`! The ASP.NET Core framework libraries in .NET 7 have fully embraced nullable reference types. See the documentation to learn more: http://mng.bz/7V0g.

For APIs, which typically are consumed by apps (as opposed to end users), that result probably is fine. But for apps that typically generate HTML, such as Razor Pages apps, returning a 404 typically results in a generic, unfriendly page being shown to the user, as you saw in figure 4.8. Although this behavior is correct, it doesn't provide a great experience for users of these types of applications.

Error-handling middleware attempts to address these problems by modifying the response before the app returns it to the user. Typically, error-handling middleware returns either details on the error that occurred or a generic but friendly HTML page to the user. You'll learn how to handle this use case in chapter 13 when you learn about generating responses with Razor Pages.

The remainder of this section looks at the two main types of exception-handling middleware that's available for use in your application. Both are available as part of the base ASP.NET Core framework, so you don't need to reference any additional NuGet packages to use them.

4.3.1 Viewing exceptions in development: DeveloperExceptionPage

When you're developing an application, you typically want access to as much information as possible when an error occurs somewhere in your app. For that reason, Microsoft provides `DeveloperExceptionPageMiddleware`, which you can add to your middleware pipeline by using

```
app.UseDeveloperExceptionPage();
```

> **NOTE** As shown previously, `WebApplication` automatically adds this middleware to your middleware pipeline when you're running in the `Development` environment, so you don't need to add it explicitly. You'll learn more about environments in chapter 10.

When an exception is thrown and propagates up the pipeline to this middleware, it's captured. Then the middleware generates a friendly HTML page, which it returns with a 500 status code, as shown in figure 4.15. This page contains a variety of details about the request and the exception, including the exception stack trace; the source code at the line the exception occurred; and details on the request, such as any cookies or headers that were sent.

Having these details available when an error occurs is invaluable for debugging a problem, but they also represent a security risk if used incorrectly. You should never return more details about your application to users than absolutely necessary, so you should use `DeveloperExceptionPage` only when developing your application. The clue is in the name!

> **WARNING** Never use the developer exception page when running in production. Doing so is a security risk, as it could publicly reveal details about your application's code, making you an easy target for attackers. `WebApplication` uses the correct behavior by default and adds the middleware only when running in development.

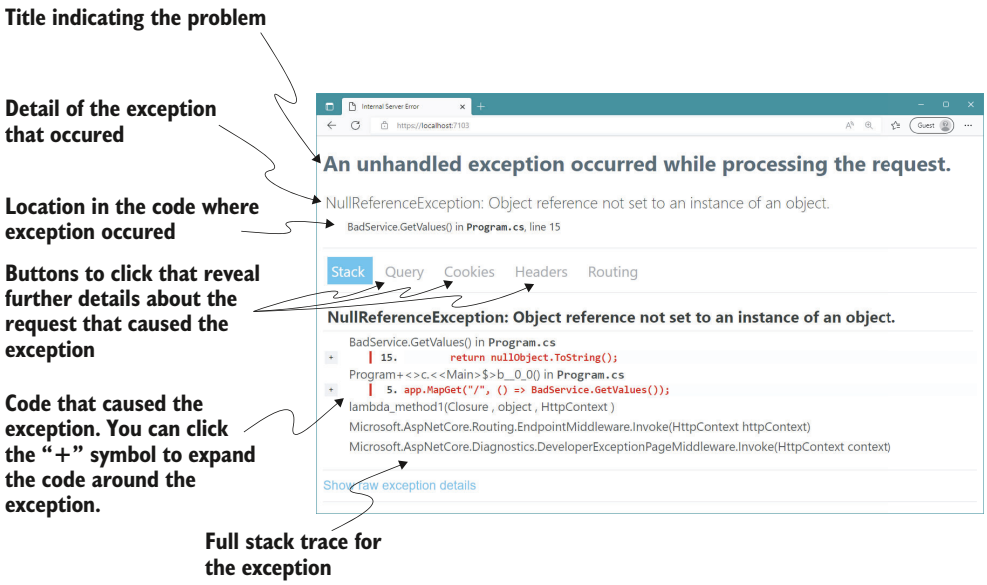


Figure 4.15 The developer exception page shows details about the exception when it occurs during the process of a request. The location in the code that caused the exception, the source code line itself, and the stack trace are all shown by default. You can also click the Query, Cookies, Headers, and Routing buttons to reveal further details about the request that caused the exception.

If the developer exception page isn't appropriate for production use, what should you use instead? Luckily, you can use another type of general-purpose error-handling middleware in production: `ExceptionHandlerMiddleware`.

4.3.2 Handling exceptions in production: ExceptionHandlerMiddleware

The developer exception page is handy when you're developing your applications, but you shouldn't use it in production, as it can leak information about your app to potential attackers. You still want to catch errors, though; otherwise, users will see unfriendly error pages or blank pages, depending on the browser they're using.

You can solve this problem by using `ExceptionHandlerMiddleware`. If an error occurs in your application, the user will see a custom error response that's consistent with the rest of the application but provides only necessary details about the error. For a minimal API application, that response could be JSON or plain text, as shown in figure 4.16.

For Razor Pages apps, you can create a custom error response, such as the one shown in figure 4.17. You maintain the look and feel of the application by using the same header, displaying the currently logged-in user, and displaying an appropriate message to the user instead of full details on the exception.

Given the differing requirements for error handlers in development and production, most ASP.NET Core apps add their error-handler middleware conditionally,

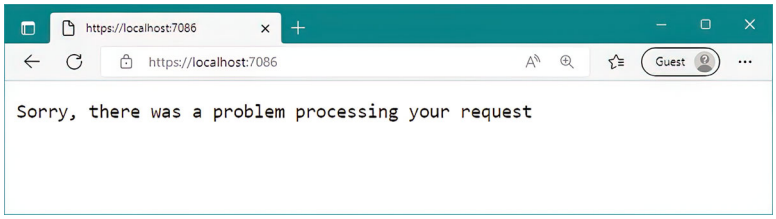


Figure 4.16 Using the `ExceptionHandlerMiddleware`, you can return a generic error message when an exception occurs, ensuring that you don't leak any sensitive details about your application in production.

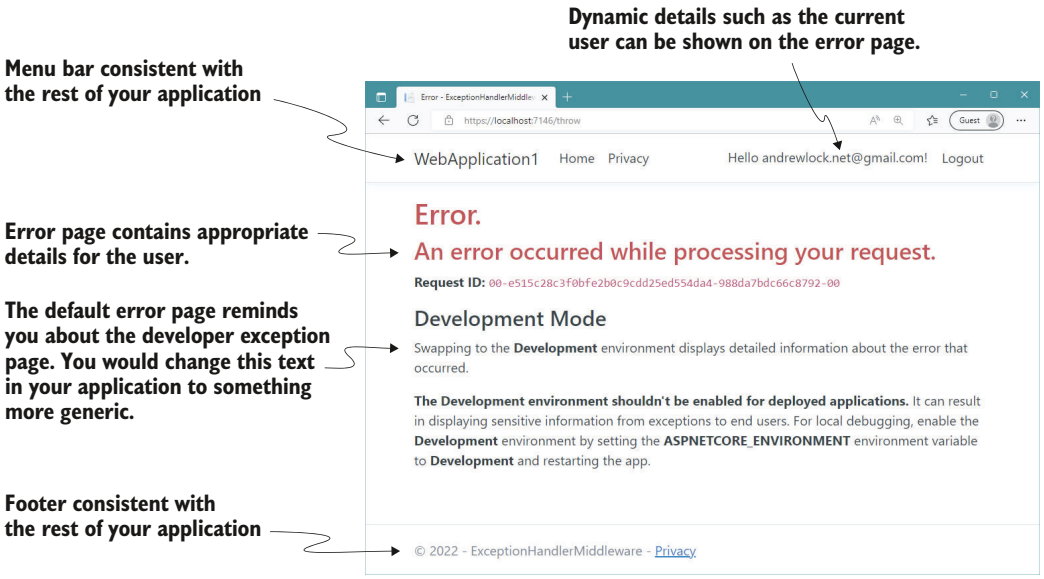


Figure 4.17 A custom error page created by `ExceptionHandlerMiddleware`. The custom error page can have the same look and feel as the rest of the application by reusing elements such as the header and footer. More important, you can easily control the error details displayed to users.

based on the hosting environment. `WebApplication` automatically adds the developer exception page when running in the development hosting environment, so you typically add `ExceptionHandlerMiddleware` when you're *not* in the development environment, as shown in the following listing.

Listing 4.5 Adding exception-handler middleware when in production

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
```

In development, WebApplication automatically adds the developer exception page middleware.

Configures a different pipeline when not running in development

```
    app.UseExceptionHandler("/error");
}

// additional middleware configuration
app.MapGet("/error", () => "Sorry, an error occurred");
```

The ExceptionHandlerMiddleware won't leak sensitive details when running in production.

This error endpoint will be executed when an exception is handled.

As well as demonstrating how to add `ExceptionHandlerMiddleware` to your middleware pipeline, this listing shows that it's perfectly acceptable to configure different middleware pipelines depending on the environment when the application starts. You could also vary your pipeline based on other values, such as settings loaded from configuration.

> **NOTE** You'll see how to use configuration values to customize the middleware pipeline in chapter 10.

When adding `ExceptionHandlerMiddleware` to your application, you typically provide a path to the custom error page that will be displayed to the user. In the example in listing 4.5, you used an error handling path of `"/error"`:

```
app.UseExceptionHandler("/error");
```

`ExceptionHandlerMiddleware` invokes this path after it captures an exception to generate the final response. The ability to generate a response dynamically is a key feature of `ExceptionHandlerMiddleware`; it allows you to reexecute a middleware pipeline to generate the response sent to the user.

Figure 4.18 shows what happens when `ExceptionHandlerMiddleware` handles an exception. It shows the flow of events when the minimal API endpoint for the `"/"` path generates an exception. The final response returns an error status code but also provides an error string, using the `"/error"` endpoint.

The sequence of events when an unhandled exception occurs somewhere in the middleware pipeline (or in an endpoint) after `ExceptionHandlerMiddleware` is as follows:

1. A piece of middleware throws an exception.
2. `ExceptionHandlerMiddleware` catches the exception.
3. Any partial response that has been defined is cleared.
4. The `ExceptionHandlerMiddleware` overwrites the request path with the provided error-handling path.
5. The middleware sends the request back down the pipeline, as though the original request had been for the error-handling path.
6. The middleware pipeline generates a new response as normal.
7. When the response gets back to `ExceptionHandlerMiddleware`, it modifies the status code to a 500 error and continues to pass the response up the pipeline to the web server.

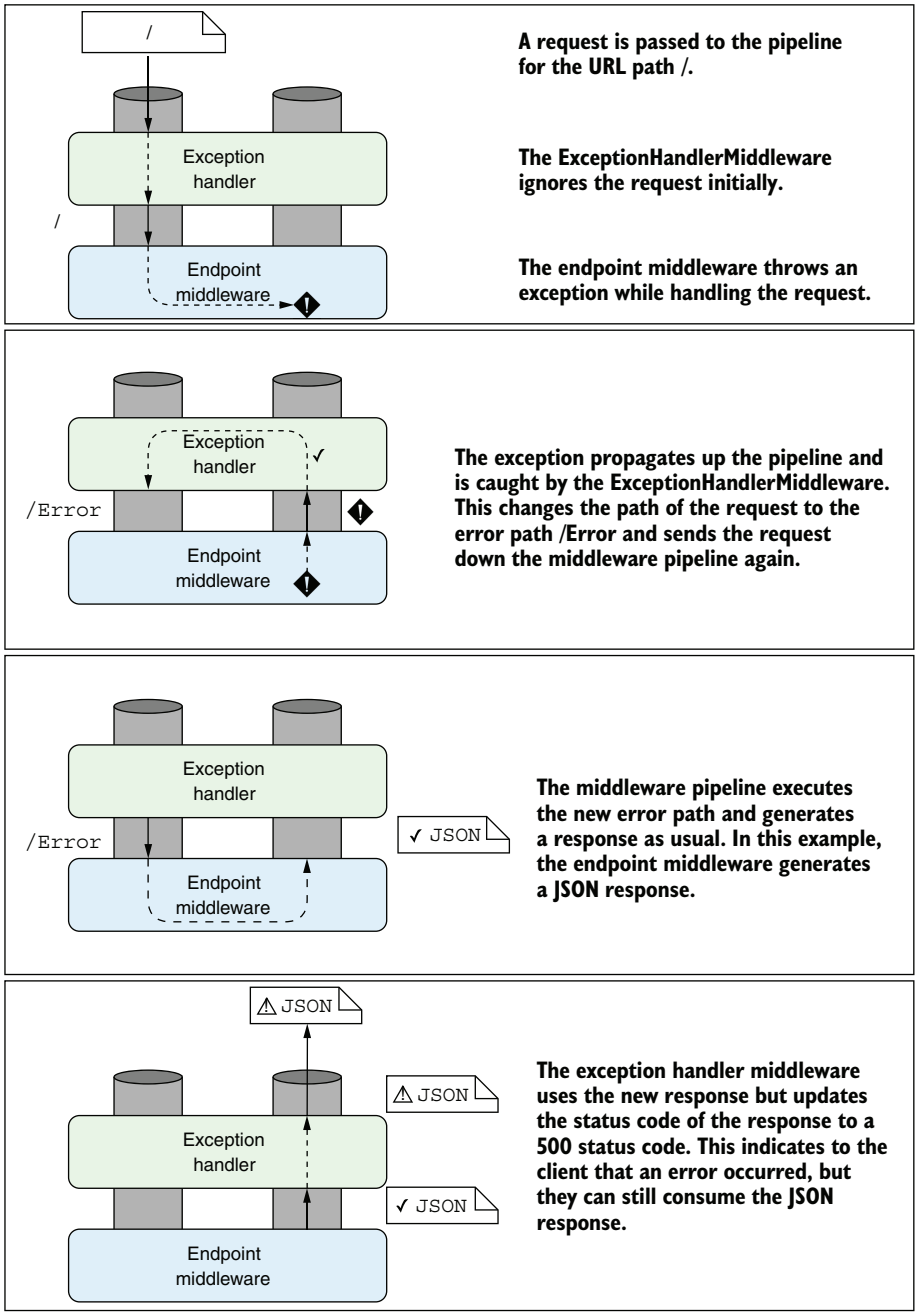


Figure 4.18 `ExceptionHandlerMiddleware` handling an exception to generate a JSON response. A request to the `/` path generates an exception, which is handled by the middleware. The pipeline is reexecuted, using the `/error` path to generate the JSON response.

One of the main advantages of reexecuting the pipeline for Razor Page apps is the ability to have your error messages integrated into your normal site layout, as shown in figure 4.17. It's certainly possible to return a fixed response when an error occurs without reexecuting the pipeline, but you wouldn't be able to have a menu bar with dynamically generated links or display the current user's name in the menu, for example. By reexecuting the pipeline, you ensure that all the dynamic areas of your application are integrated correctly, as though the page were a standard page of your site.

> **NOTE** You don't need to do anything other than add `ExceptionHandlerMiddleware` to your application and configure a valid error-handling path to enable reexecuting the pipeline, as shown in figure 4.18. The middleware will catch the exception and reexecute the pipeline for you. Subsequent middleware will treat the reexecution as a new request, but previous middleware in the pipeline won't be aware that anything unusual happened.

Reexecuting the middleware pipeline is a great way to keep consistency in your web application for error pages, but you should be aware of some gotchas. First, middleware can modify a response generated farther down the pipeline only if the response *hasn't yet been sent to the client.* This situation can be a problem if, for example, an error occurs while ASP.NET Core is sending a static file to a client. In that case, ASP.NET Core may start streaming bytes to the client immediately for performance reasons. When that happens, the error-handling middleware won't be able to run, as it can't reset the response. Generally speaking, you can't do much about this problem, but it's something to be aware of.

A more common problem occurs when the error-handling path throws an error during the reexecution of the pipeline. Imagine that there's a bug in the code that generates the menu at the top of the page in a Razor Pages app:

1. When the user reaches your home page, the code for generating the menu bar throws an exception.
2. The exception propagates up the middleware pipeline.
3. When reached, `ExceptionHandlerMiddleware` captures it, and the pipe is reexecuted, using the error-handling path.
4. When the error page executes, it attempts to generate the menu bar for your app, which again throws an exception.
5. The exception propagates up the middleware pipeline.
6. `ExceptionHandlerMiddleware` has already tried to intercept a request, so it lets the error propagate all the way to the top of the middleware pipeline.
7. The web server returns a raw 500 error, as though there were no error-handling middleware at all.

Thanks to this problem, it's often good practice to make your error-handling pages as simple as possible to reduce the possibility that errors will occur.

> **WARNING** If your error-handling path generates an error, the user will see a generic browser error. It's often better to use a static error page that always works than a dynamic page that risks throwing more errors. You can see an alternative approach using a custom error handling function in this post: http://mng.bz/0Kmx.

Another consideration when building minimal API applications is that you generally don't want to return HTML. Returning an HTML page to an application that's expecting JSON could easily break it. Instead, the HTTP 500 status code and a JSON body describing the error are more useful to a consuming application. Luckily, ASP.NET Core allows you to do exactly this when you create minimal APIs and web API controllers.

> **NOTE** I discuss how to add this functionality with minimal APIs in chapter 5 and with web APIs in chapter 20.

That brings us to the end of middleware in ASP.NET Core for now. You've seen how to use and compose middleware to form a pipeline, as well as how to handle exceptions in your application. This information will get you a long way when you start building your first ASP.NET Core applications. Later, you'll learn how to build your own custom middleware, as well as how to perform complex operations on the middleware pipeline, such as forking it in response to specific requests. In chapter 5, you'll look in depth at minimal APIs and at how they can be used to build JSON APIs.

Summary

- Middleware has a similar role to HTTP modules and handlers in ASP.NET but is easier to reason about.
- Middleware is composed in a pipeline, with the output of one middleware passing to the input of the next.
- The middleware pipeline is two-way: requests pass through each middleware on the way in, and responses pass back through in reverse order on the way out.
- Middleware can short-circuit the pipeline by handling a request and returning a response, or it can pass the request on to the next middleware in the pipeline.
- Middleware can modify a request by adding data to or changing the `HttpContext` object.
- If an earlier middleware short-circuits the pipeline, not all middleware will execute for all requests.
- If a request isn't handled, the middleware pipeline returns a 404 status code.
- The order in which middleware is added to `WebApplication` defines the order in which middleware will execute in the pipeline.
- The middleware pipeline can be reexecuted as long as a response's headers haven't been sent.

- When it's added to a middleware pipeline, `StaticFileMiddleware` serves any requested files found in the wwwroot folder of your application.
- `DeveloperExceptionPageMiddleware` provides a lot of information about errors during development, but it should never be used in production.
- `ExceptionHandlerMiddleware` lets you provide user-friendly custom error-handling messages when an exception occurs in the pipeline. It's safe for use in production, as it doesn't expose sensitive details about your application.
- Microsoft provides some common middleware, and many third-party options are available on NuGet and GitHub.